Abstract

This paper presents the first entropy estimator for DNA sequences that has both proven properties and excellent entropy estimates. Additionally, our algorithm, the Grammar Transform Analysis and Compression (GTAC) entropy estimator, uses a novel data structure to repeatedly solve the longest non-overlapping pattern problem in linear time. GTAC beats all known competitors in running time, in the low values of its entropy estimates, and in the properties that have been proven about it.

1 Introduction

With the complete DNA sequences of many organisms already known, and the goal of completely sequencing the human genome making steady progress, the challenge of using this wealth of genetic information beneficially presents a new series of problems to be solved. One such problem is recognizing and characterizing the different regions of DNA and their different functions. Information theoretic entropy is one tool that can be used to help solve these types of problems.

DNA stores the information for creating and organizing an organism. It can be thought of as a string over the alphabet {A, C, G, T}, which represents the four chemical components that make it up. In order to synthesize protein, the cell creates a transient copy of a portion of DNA, called mRNA. The sequence of mRNA is read, three symbols at a time, and this triplet specifies a single component of the protein called an amino acid. The components are strung together in the order their code appears in the mRNA sequence. Although made as a linear structure, many types of protein can fold up on themselves and become active structures that control or influence a chemical reaction in the cell.

No all of the sequence information that gets copied from the DNA ends up specifying a protein. In higher eukaryotes (such as plants and animals) much of the mRNA is cut out before the cell translates it into protein. The portions that are translated are called exons and the portions that are removed are called introns. Random changes in a sequence are thought to be more deleterious if they take place in an exon rather than in an intron so these two regions should have different information theoretic entropy.

Since the alphabet of DNA, and mRNA have four symbols, if these sequences were totally random, it would take two bits per symbol to represent the sequence. However, only a small fraction of DNA sequences result in a viable organism, therefore there are constraints on those sequences which appear in a living organism. Common compression algorithms, such as Huffman, and various Lempel-Ziv based algorithms fail to compress DNA sequences at all (for an example using UNIX compress, see Table 2), and low order arithmetic encoding algorithms only compress to around 1.96 bits per base [4]. With only 61 of the 64 possible mRNA triplet combinations being used to code for protein, that property alone suggests an encoding of 1.977 bits per symbol for exons, so the arithmetic encoding is providing only a slight improvement.

2 Previous Work

There have been several previous attempts to characterize the entropy of DNA. One of the most common approaches is to estimate the probability of $n$-tuples for large $n$, and use this value to compute the block entropy. The problem with this approach is that it converges too slowly, so even though genome databases are large and growing larger, the values that are obtained systematically underestimate the entropy due to the finite sample effect, and must be corrected. Several researchers address this problem and have developed methods to correct it, such as Li et al. [10], and Schmitt and Herzel [13].
2.1 Biocompress Another approach is to compress the sequence in order to determine an upper bound on the entropy. Such a method has been used by Grumbach and Tahi with their two algorithms Biocompress [3] and Biocompress-2 [4]. Biocompress-2 is actually a combination of three approaches: 1) Literal encoding, where each symbol is coded as a two bit number; 2) Arithmetic encoding, where a substring of symbols is encoded using second order arithmetic encoding; 3) Lempel-Ziv style encoding, where a substring is encoded as a pair of integers, one representing the length of the match, and the second representing the position of the match somewhere to the left of the current position. Biocompress-2 checks to see which method is more efficient to encode a small portion of the input, and then encodes the type of encoding, the length, followed by the actual encoding. The entropy estimates provided by Biocompress-2 are presented in Table 2.

2.2 Match Length Entropy Estimator Farach et al. [2] developed a novel algorithm to estimate the entropy of DNA sequences called a match length entropy estimator. Letting \( L_i \) represent the match length at the \( i^{th} \) character in a sequence, the value of \( L_i \) is the length of the longest substring that occurs in two places: 1) starting at position \( i + 1 \), and 2) a sliding window consisting of the previous \( N_w \) characters. This algorithm was used to test the differences between the entropy of introns and exons, and contrary to what was expected, they found that the average entropy of exons was larger 73% of the time and that the variability of introns was larger 80% of the time. Farach et al. also proved that their algorithm was universal, that is, that the entropy estimate will approach the true entropy as the size of the sequence increases, but only under the assumption that the sequence is generated by a Markov process.

2.3 CDNA Loewenstern and Yianilos [11, 17] developed CDNA, a program that estimates the entropy of DNA sequences. The motivation for CDNA comes from the observation that naturally occurring DNA sequences contain many more near repeats then would be expected by chance. Two parameters that CDNA uses to capture the inexact matches are \( w \), which represents the substring size, and \( h \), which represents the Hamming distance. These parameters are used to create a panel of predictive experts \( p_{w,h} \), each with differing values of \( w \) and \( h \). CDNA then learns the weightings of these various experts, using Expectation Maximization, so that their predictive ability is maximized when combined into a single prediction.

CDNA has been implemented in two different ways. In the cross validation approach, the sequence is partitioned into 20 equal segments, the algorithm is trained on 19 of the segments and predicts the remaining segment. This approach is repeated 20 times using a different segment as the test segment each time. The value reported is the average of the 20 iterations.

While the cross validation estimate is used to remove the overhead associated with compression, a simple example will illustrate how this approach can severely underestimate the entropy of a genetic sequence. Let the input be a tandem repeat, say \( r^r \), where \( r \) is a random genetic sequence and hence has an entropy of two bits per symbol. The cross validation approach will report an entropy of close to zero, yet the entropy of \( r^r \) is one bit per base. How likely is such an example? Both tandem repeats, such as the one above, and dispersed repeats are known to occur in DNA and comprise a substantial fraction of the human genome [9].

The second method that Loewenstern and Yianilos use is called CDNA compress, in which the algorithm uses everything to the left of a nucleotide as the learning set to predict its value. The average over all positions is calculated and that is the value that is recorded in Table 2.

2.4 Our Results In this paper, we develop a novel entropy estimator of DNA sequences, GTAC, which is based on the idea of Kieffer and Yang [7] regarding the design and analysis of grammar based codes, and which recognizes the reverse complement property of DNA sequences. Our entropy estimator is universal in the sense that it does not assume any source model and works for any individual sequence. Moreover, our entropy estimator is well justified from the information theoretic point of view.

In Table 1, we compare GTAC with the best known entropy estimators using three criteria:

- Is the code universal with respect to any stationary source? That is, will the entropy estimate converge to the actual entropy if the sequence is long enough? A limited one, such as the Match Length entropy estimator must make the addition assumption that the source is a Markov process.
- Is the run time linear?
- How good are the algorithm's entropy estimates?

In summary, algorithms such as Unix compress and Biocompress-2 were designed as compression algorithms, so they tend to overestimate the entropy because they include overhead necessary for compression. CDNA was designed as an entropy estimator, but no
convergence properties have been proven about it and its entropy estimates are inferior to GTAC’s.

3 Our Approach

Before presenting our entropy estimator, we first briefly review grammar based codes, and then discuss how well the corresponding entropy estimators are.

3.1 Grammar Based Codes

A context-free grammar (CFG) is a quadruple $G = (V, T, P, S)$ where $V$ is a finite non-empty set of variables, $T$ is a finite non-empty set of terminal symbols that is disjoint from $V$, $S$ is a distinguished element of $V$ called the start symbol, and $P$ is a set of production rules which map elements of $V$ onto $(V \cup T)^*$. Using the concept of CFG, Kiefer and Yang [7, 15, 16] recently put forth a new type of lossless source code called a grammar based code, and developed a new universal lossless source coding theory. In this theory, a grammar based code has the structure shown in Figure 1. The original data sequence $x$ is first transformed into a context-free grammar (or simply a grammar) $G_x$, from which $x$ can be fully recovered, and then compressed indirectly by using a zero order arithmetic code to compress $G_x$. To get an appropriate $G_x$, string matching is often used in some manner.

Since one has to fully recover $x$ from $G_x$, it is required that $G_x$ satisfy the following constraints:

- The language generated by $G_x$ consists of only $x$.
- $G_x$ is deterministic, that is, any variable in $V$ appears only once on the left hand side of the production rules, $P$.
- $P$ does not contain the empty string on the right hand side of any rule.
- $G_x$ has no useless symbols. That is, during the process of deriving $x$ from the production rule corresponding to the start symbol $S$, each production rule in $G_x$ is used at least once.

Such a grammar is called an admissible grammar.

**Example 1:** Below is an admissible grammar $G_x$ with $x = aataaatgcataatatgce$.

A grammar transform is a process of deriving a grammar from some data. For the CF grammar $G_x$ shown in Example 1, $\omega(G_x) = BC DDaaAtatgCgc$ and $H(G_x) = 34.26$. The following theorem, proved in [7, 15], characterizes the resulting compression rate.

**Theorem 3.1.** According to arithmetic coding or enumerative coding, one can assign a uniquely decodable binary codeword $B(G_x)$ to each admissible CFG $G_x$ (or its equivalent form) such that

$$\omega(G_x) = \sum_s n(s) \log \omega(G_x)$$

where $n(s)$ denotes the number of times the variable (or terminal symbol) $s$ appears in $\omega(G_x)$, and the logarithm is relative to base 2, and the convention $0 \log \infty = 0$ is adopted. In terms of the terminology in [7], the quantity $H(G_x)$ is called the unnormalized entropy of the grammar $G_x$. For the CFG $G_x$ shown in Example 1, $\omega(G_x) = BC DDaaAtatgCgc$ and $H(G_x) = 34.26$. The following theorem, proved in [7, 15], characterizes the resulting compression rate.

$$\omega(G_x) = \sum_s n(s) \log \omega(G_x)$$
negligible as compared to \( H(G_x) \) and is upper bounded, in the worst case scenario, by
\[
f(G_x) \leq 5|G_x| + \alpha
\]
where \(|G_x|\) denotes the total entries in the right hand side of all production rules of \( G_x \), and \( \alpha \) is the cardinality of the source alphabet and is 4 in the case of DNA sequences.

From Theorem 3.1, it follows that in order for a grammar based code to be efficient, the corresponding grammar transform \( x \rightarrow G_x \) should be designed so that the unnormalized entropy \( H(G_x) \) of \( G_x \) and the overhead \( f(G_x) \) are as small as possible. An interesting type of grammar transform called irreducible grammar transform has been identified in [7]. An admissible grammar \( G_x \) is said to be irreducible if the following properties hold:

\( \textbf{P1} \) Every variable of \( G_x \) other than the start symbol \( S \) appears at least twice in the right hand side of all production rules of \( G_x \).

\( \textbf{P2} \) There is no repeated pattern of length \( \geq 2 \) in the right hand side of all production rules of \( G_x \).

\( \textbf{P3} \) Each distinct variable of \( G_x \) represents a distinct substring of \( x \).

The admissible grammar \( G_x \) shown in Example 1 is irreducible. A grammar transform is said to be irreducible if it converts every sequence \( x \) into an irreducible grammar \( G_x \). Several reduction rules have also been proposed in [7] to reduce any reducible grammar transforms into irreducible ones. Starting with the grammar consisting of only one production rule \( S \rightarrow x \) and applying repeatedly these reduction rules in different orders, one can design, in principle, many irreducible grammar transforms. Among them are the longest matching substring grammar transform [7], which will be analyzed and extended to DNA sequences, and the Yang-Kiefer greedy sequential grammar transform [15], which gives rise to universal data compression algorithms significantly outperforming the Lempel-Ziv type of algorithms such as the Unix Compress and Gzip algorithms. Interestingly enough, no matter how one gets irreducible grammar transforms, they all give rise to efficient universal compression algorithms, as shown in the following theorem (see [7] and [8] for its proof).

\[
\text{Theorem 3.2. For any sequence } x, \text{ let } \mathcal{G}(x) \text{ be the set of all irreducible grammars } G \text{ representing } x. \text{ Then the following hold.}
\]

(a) There is a constant \( c \), which depends only on the cardinality of the source alphabet, such that for any sequence \( x \)
\[
\max_{G \in \mathcal{G}(x)} |G| \leq c \frac{|x|}{\log |x|}
\]

where \(|x|\) denotes the length of \( x \).

\( \text{b) For any stationary, ergodic source } \{X_i\}_{i=1}^{\infty} \text{ with } \text{entropy } H, \text{ the quantity}
\]
\[
\max \left\{ \frac{|B(G)|}{n} - H : G \in \mathcal{G}(X_1 \cdots X_n) \right\}
\]

\( \text{goes to } 0 \text{ with probability one as } n \rightarrow \infty. \)

\( \text{Remark 1. Part (b) of Theorem 3.2 represents the worst case scenario. The actual convergence rate at which}
\]
\[
\frac{|B(G_{x^n})|}{n} - H
\]

\( \text{where } G_{x^n} \text{ is an irreducible grammar representing } X^n = X_1 \cdots X_n, \text{ goes to } 0 \text{ depends on the source}
\]
\( \{X_i\}_{i=0}^{\infty} \text{ and the irreducible grammar transform } X^n \rightarrow G_{x^n}. \text{ Also, from Theorems 3.1 and 3.2, it follows that for any irreducible grammar transform } x \rightarrow G_x, \text{ the normalized grammar entropy}
\]
\( H(G_{x^n})/n \text{ goes to the actual entropy } H \text{ with probability one as } n \rightarrow \infty. \)

3.2 Grammar Based Codes Entropy Estimators

As suggested by Theorem 3.2 and Remark 1, we can associate an entropy estimator with any grammar based code. Given a grammar based code with grammar transform \( x \rightarrow G_x \), we define an entropy estimator which simply uses the normalized grammar entropy \( H(G_x)/|x| \text{ of } G_x \text{ as an estimate to the entropy in bits per letter of } x \). The reason we do not include the normalized overhead \( f(G_x)/|x| \) in the definition of the associated entropy estimator is obvious—the inclusion generally results in an overestimate. When the underlying grammar transform satisfies Property \( \textbf{P2} \) and is asymptotically compact in the sense that \( |G_x|/|x| \text{ goes to } 0 \text{ as } |x| \rightarrow \infty \), some ergodic behavior of such entropy estimators has been investigated in [8]. In this paper, we focus on a more restricted case in which the underlying grammar transform is irreducible.

In terms of the terminology in estimation theory, Theorem 3.1, Theorem 3.2, and Remark 1 imply that whenever the underlying grammar transform is irreducible, the corresponding associated entropy estimator is consistent and asymptotically unbiased for the class of stationary, ergodic sources. Although it would be nice to determine the convergence rate at which the estimate provided by such an entropy estimator goes to the actual entropy \( H \), this problem is in general very difficult and thus left open for future research. Furthermore, as indicated in Remark 1, the solution to this problem depends on the actual source model and the underlying irreducible grammar transform. In our case of DNA sequences, we have no idea what kind of source model DNA sequences would obey. To get around these
difficulties, in the following, we shall take a different approach. Instead, we shall show that no matter what the underlying irreducible grammar transform and the source model are, with a high probability these entropy estimators will never severely underestimate the actual entropy.

**Theorem 3.3.** Let \( \{X_i\}_{i=1}^n \) be any data source. Then for a constant \( d > 0 \), the following holds with probability at least \( 1 - n^{-d} \):

\[
\frac{|H(G_x)|}{n} \geq -\frac{1}{n} \log P(X^n) - f(G_x) - d \log n
\]

for any grammar transform \( x \rightarrow G_x \), where \( X^n = X_1 \cdots X_n \) and \( P(X^n) \) denotes the probability of \( X^n \).

**Remark 2.** From information theory, one can interpret \( (-\log P(X^n))/n \) as the entropy in bits per letter of \( X^n \). From Theorems 3.1 and 3.2, it follows that for irreducible grammar transforms, \( f(G_x)/n \) is quite small and upper bounded, in the worst case scenario, by \( O(1/\log n) \). Therefore, Theorem 3.3 says that with a high probability, the entropy estimators associated with grammar based codes with irreducible grammar transforms will never severely underestimate the actual entropy.

**Proof of Theorem 3.3:** Let \( F_n \) denote the set consisting of all sequences \( x \) of length \( n \) for which there is a grammar transform \( x \rightarrow G_x \) such that

\[
H(G_x) \leq -\log P(x) - f(G_x) - d \log n
\]
or equivalently,

\[
\log P(x) \leq -H(G_x) - f(G_x) - d \log n
\]

Then we have

\[
\Pr \{X^n \in F_n\} = \sum_{x \in F_n} P(x) \\
\leq \sum_{x \in F_n} 2^{-H(G_x) - f(G_x) - d \log n} \\
= n^{-d} \sum_{x \in F_n} 2^{-|B(G_x)|} \\
\leq n^{-d}
\]

where the last inequality is due to the fact that the binary codeword \( B(G_x) \) is uniquely decodable, and hence the Kraft inequality among all admissible grammars holds:

\[
\sum_{G \text{ is admissible}} 2^{-|B(G)|} \leq 1
\]

From (3.3), Theorem 3.3 follows.

Theorem 3.3 suggests that among all entropy estimators associated with grammar based codes with irreducible grammar transforms, the best one correspond to the irreducible grammar transform which gives the least grammar entropy \( H(G_x) \). However, we believe that finding the irreducible grammar with the least grammar entropy \( H(G_x) \) is an NP-hard problem. Instead, in the following section, we shall present a linear time algorithm for constructing an irreducible grammar with a good entropy estimate.

4 The Algorithm

The Grammar Transform Analysis and Compression (or GTAC) is an example of a Grammar Based Code. The core of GTAC is to repeatedly solve the longest non-overlapping pattern (LNP) problem. The LNP problem is as follows: Given a set of strings \( \mathcal{P} \), find the longest substring \( \beta \) such that \( \beta \) occurs in at least two non-overlapping positions somewhere in \( \mathcal{P} \). The LNP problems can appear in the context of grammar, \( G = (V, T, P, S) \), when we let \( \mathcal{P} \) be the set of all right hand sides of the production rules \( P \), and we add the additional constraint that the length of \( \beta \) is at least two. GTAC’s goal is to repeatedly find the LNP and reduce it, creating a new rule. If an LNP \( \beta \) appears in the following form (both in the same string),

\[
A \rightarrow \alpha_1 * \beta * \alpha_2 * \beta * \alpha_3
\]

rewrite the previous rule as two rules,

\[
A \rightarrow \alpha_1 * B * \alpha_2 * B * \alpha_3
\]

\[
B \rightarrow \beta
\]

If an LNP \( \beta \) appears in different rules,

\[
A \rightarrow \alpha_1 * \beta * \alpha_2
\]

\[
B \rightarrow \alpha_3 * \beta * \alpha_4
\]

then rewrite the previous rules and introduce a new one, as follows.

\[
A \rightarrow \alpha_1 * C * \alpha_2
\]

\[
B \rightarrow \alpha_3 * C * \alpha_4
\]

\[
C \rightarrow \beta
\]

GTAC can optionally recognize reverse complements. In DNA sequences, the symbols \( a \) and \( t \) are the complement of each other, and the symbols \( g \) and \( c \) are the complement each other. A string \( \beta' \) is the reverse complement of \( \beta \) if \( \beta' \) is the reverse of \( \beta \) with each character complemented. For example, the reverse complement of \( aacgt \) is \( acgttt \). As Section 5 makes
clear, the ability to detect reverse complements is an important feature of a DNA entropy estimator.

The GTAC algorithm deals with reverse complements by having two sets of non-terminals, regular ones $A_1, A_2, \ldots$ and reverse complement ones $R_1, R_2, \ldots$. These non-terminals come into play as follows. Given an input, $x$, the algorithm first creates the trivial grammar $S \rightarrow x$. Next GTAC finds the LNP. If there is two or more occurrences of $\beta$, create a rule $A_i \rightarrow \beta$ ignoring any occurrences of $\tilde{\beta}$. If there is only one occurrence of $\beta$ and one of $\tilde{\beta}$, then create a rule using one of the reverse complement non-terminals, $R_i \rightarrow \beta$, which means interpret the second occurrence of the non-terminal in the right hand side of the rule as the reverse complement of the rule.

For example, given the input aatactgagtaaa, GTAC first creates the trivial grammar.

$$S \rightarrow \text{aatactgagtaaa}$$

Next GTAC finds the largest LNP, which is tact and its reverse complement aagt. GTAC reduces this substring and creates a new rule

$$S \rightarrow \text{a} A_0 R_0 \text{g} R_0 \text{a} a$$

and

$$R_0 \rightarrow \text{tact}$$

Next, any remaining LNPs are rewritten, in this case there is one, aa.

$$S \rightarrow \text{A}_0 R_0 g R_0 A_0$$

and

$$R_0 \rightarrow \text{tact}$$

and

$$A_0 \rightarrow \text{aa}$$

Next, relabel the non-terminals in order of appearance with the reverse complement rules starting with symbol $R_0$, and the normal rules with symbol $A_0$.

$$A_0 \rightarrow \text{A}_1 R_0 g R_0 A_1$$

and

$$R_0 \rightarrow \text{tact}$$

and

$$A_1 \rightarrow \text{aa}$$

The right hand sides of the rules are concatenated together in the following order: the start rule is first, followed by any reverse complement rules, followed by any normal rules and the first occurrence of each non-terminal is deleted.

$$g R_0 A_1 \text{aatact}$$

The entropy is then calculated based on the frequency of appearance of each symbol using Equation 3.1.

Trivial implementations of this algorithm requires $O(n^3)$ time, where $n$ is often in the order of a million or more. For this size, even an $O(n^2)$ algorithm becomes intolerable. Since a key feature of this approach is to repeatedly look for the LNP, the generalized suffix tree (a suffix tree that contains more than one string) is a natural data structure to consider because it can find the LNP in time linear in the total length of the right hand side of the grammar. However, GTAC continually rewrites the rules, reducing the size of the grammar, so a key challenge is keeping the suffix tree up-to-date.

Consider the following example where catactagt is a substring in the input with cat, tact and tag appearing elsewhere in the input. When an LNP is discovered, in this case tact, rewriting it not only affects the LNP, it also affects any patterns that overlap with the LNP, such as cat and tag. If the algorithm finds an LNP that occurs $n$ times, and is $l$ characters long, then rewriting it can affect $O(nl)$ other suffixes.

Another complicating factor is that a generalized suffix tree directly reports the longest pattern (LP), but our algorithm requires the longest non-overlapping pattern, hence we must be able to obtain one from the other. The following lemma gives an algorithm that obtains this result, and provides a bound for the size of the LNP given the LP.

**Lemma 4.1.** If the length of the LP is $l$, then an LNP can be found with length at least $\lceil l/2 \rceil$.

**Proof.** Let the reported LP start at positions $k$ and $k+i$. If $i \geq \lceil l/2 \rceil$ we are done. If not, then the LP is a periodic string, and one can find the start of the second string at or beyond $k+\lceil l/2 \rceil$ and it will be at least $\lceil l/2 \rceil$ characters long. □

The preceding lemma characterizes the situation when the LP occurs twice, and the next lemma is needed when the LP occurs three or more times.

**Lemma 4.2.** Given an LP that occurs three times in a string, at least two of those occurrences do not overlap with each other.

**Proof.** Proof by contradiction. Assume that any two of the three substrings overlap with each other. Let the substrings start at $k$, $k+i$, and $k+j$ with $0 < i < j < l$, where $l$ is the length of the LP. The LP is periodic, and one can use that property with the substring starting at $k+j$ to show that the other two substrings match at $k+l$ and $k+l+i$ contradicting the fact that they are LPs. □

With these two lemmas a subroutine for dealing with overlapping matches can be outlined. If an LP has
just two occurrences, check for overlap, and if necessary create non-overlapping substrings. Given three or more matches, that LP will also be an LNP, so keep it as is.

4.1 Observations The data structure for GTAC is a suffix tree, along with a copy of the original input, and an array of queues. In order to understand how they interrelate, a few observations are necessary first.

Observation 1. Since GTAC always considers the longest pattern at each iteration, if it is currently looking at an LNP of length \( l \), the longest branch in the tree is at most \( 2l \) long, reflecting the fact the tree may contain a length \( 2l \) LP that corresponds to a length \( l \) LNP. Hence the most number of characters that the algorithm will have to search down the tree before uniquely identifying a suffix is \( 2l + 1 \), because at that point it is guaranteed to be in a leaf. So if the LNP starts at position \( k \), only suffixes in the range \([k - 2l, k + l - 1]\) needs to be checked to see if they contains a pattern that overlaps the LNP.

Observation 2. No rule will contain a non-terminal from a previous iteration. For example, if a substring \( \beta_1 \beta_2 \ldots \beta_k \) being replaced by a non-terminal \( A_i \), then there will never be a pattern \( cA \) or \( A_i c \), for some \( c \), found later on in the grammar, because \( \beta_1 \beta_2 \ldots \beta_k \) was the longest pattern at that point of time. Since an LNP will never straddle over the beginning of non-terminal after it’s been introduced, the suffix can be considered to end at that point in the suffix tree. For example, if a path from the root is \( p_1 p_2 p_3 \beta_1 \beta_2 \ldots \), then that path could be edited as \( p_1 p_2 p_3 A_i \) or simply as \( p_1 p_2 p_3 \$. For convenience, this latter approach will be followed. However the rewrite from \( \beta_1 \beta_2 \ldots \beta_k \) to \( A_i \) has to be reflected somewhere, so the original input string is rewritten, rather than the suffix tree.

4.2 A Linear Time Algorithm With the above observations in mind, a more relaxed version of the generalized suffix tree can be used to implement the GTAC algorithm. In all, GTAC uses three data structures: a generalized suffix tree \( T \), which provides information about the LPs, an array of queues \( Q \) which bucket sorts the LNP information by size, and an array, \( x \), holding the original input which provides information about the substrings that occur to the left of a given LNP.

First \( T \) is built from the original input, and then as it is traversed information about the LNP are stored in \( Q \). Then the LNP are read from \( Q \) in descending order of size, and \( Q \) and \( T \) are kept up-to-date as the LNP get removed. The whole algorithm is outlined and explained below.

**GTAC**

1. **Initialize Data Structures**: Given an input string, \( x \), create the trivial rule \( S \to x \); a generalized suffix tree \( T \) on \( x \), and an array of queues \( Q \), with \( Q[i] \) a list of LNP of size \( i \).

2. **Fill Q**: Traverse the tree, \( T \), keeping track of the depth. At each interior node \( n \), which represents a pattern, check to see if it also represents an LNP and if so create a pointer to that node, and add it to the queue that corresponds to it’s depth, namely \( Q[\text{depth}(n)] \). Also include a back-pointer from the interior node to its corresponding entry in the queue.

3. **Get pβ**: Work through the \( Q \) array starting at the largest value \( md \). Given an LNP, say \( \beta \), from \( Q[i] \), for each occurrence of \( \beta \), consider the substring that extends up to \( 2l \) characters on either side of \( \beta \), namely \( p_1 p_2 \ldots p_i \beta \beta_1 \beta_2 \ldots \beta_k s_1 s_2 \ldots s_m \) where \( p \) represents the prefix, and \( s \) the suffix of \( \beta \). This substring can be determined by consulting \( T \) to get the list of locations of \( \beta \), and then consulting the input string \( x \) to get \( p \) and \( s \) for that occurrence.
4. Find Suffixes: For each suffix starting at $p_1 p_2 \ldots$ and ending at $\beta s_1 s_2 \ldots$, perform Steps 4 to 7. Descend down the tree on the path $p_1 p_2 \ldots$ with two pointers, $d$-ptr and $i$-ptr. The $d$-ptr will point to the leaf that corresponds to the unique suffix we are looking for and will eventually delete; the $i$-ptr will point to the node where we insert the end-of-string marker (which is always the node between $p_{l-1}$ and $\beta$). Search down this path for the beginning of the LNP, $\beta_1$. Consistent with Observation 1 above, a search will only go as far as the first character in the leaf.

5. Remove Suffixes Starting in $p_2$: If, while searching down a path, $\beta_1$ is encountered, then the algorithm will begin modifying the tree. First the $i$-ptr stays at the node between $p_{l-1}$ and $\beta_1$. A node may have to be created here and a corresponding suffix link and entry made in the $Q$ array. The $d$-ptr continues down the path to the leaf corresponding to this unique suffix. If this leaf has more than one sibling, then just delete the leaf that $d$-ptr points to. If the parent node only has only two child leaves, then delete the two leaves, and convert the parent node to a leaf corresponding to the sibling of the leaf that is being deleted. For example, if the path into the parent node is labeled $\beta_i \beta_{i+1}$ and the two leaves are labeled $\beta_{i+2} \ldots$ and $c \ldots$ then the parent node becomes the leaf corresponding to the suffix $\beta_i \beta_{i+1}c \ldots$. Wherever a node is deleted, the back-pointer from this node are followed, if it exists, and its corresponding entry in the $Q$ array is removed. As well, the end-of-string marker, $\$, is added to where the $i$-ptr points to (representing the new place where this suffix ends, as explained in Observation 2). When finished with the current suffix and moving to the next one, suffix links are used. A suffix link is a pointer from the interior node that represents the suffix starting with $\alpha$ to the one representing the suffix starting with $\beta$, where $c$ is a single character and $\alpha$ is a possibly empty string. Both the $i$-ptr and the $d$-ptr independently take the suffix links from their current node to move on to the next suffix, or go up a most one node, and use that suffix link instead.

6. Remove Suffixes Starting in $\beta$: A similar process is followed for the suffixes that start in $\beta$, except that the entire suffix is eliminated with no need to insert a end-of-string marker anywhere. If the path is $\beta_i \beta_{i+1} \ldots \beta_j s_1 s_2 \ldots s_j$, then the leaf corresponding to that suffix is eliminated from the tree entirely, and if necessary the parent node of that leaf becomes a leaf, and the corresponding entry in $Q$ eliminated. The final suffix to be considered is $\beta s_1 s_2 \ldots$.

7. Edit Rules: Finally the rule containing $\beta$ is updated by deleting that occurrence of $\beta$ and adding the appropriate non-terminal in its place.

8. Create New Rule: A new rule is created and the right hand side of that rule, $\beta$ is added to the suffix tree (actually the last occurrence of $\beta$ in the tree is converted to this new entry).

With only a few modifications this algorithm also deals with reverse complements. In the first step both the string $x$ and $x^r$ are added to the suffix tree, and when removing a suffix both the forward and the reverse complement occurrences must be removed. As well, two sets of non-terminals (for normal and reverse complement patterns) with the decision about which to use take place between steps 3 and 4. The source code for this algorithm will be available at [1].

With a description of the algorithm complete the next step is to characterize the running time, which is as follows.

**Theorem 4.1.** The GTAC entropy estimator runs in time linear in the size of its input.

**Proof.** Assume the the input has $m$ characters in it. In step 1, each statement can be performed in linear time, such as building the suffix tree [14]. Step 2 is to build and maintain $Q$, the array of queues. There are two aspects, the size of the array, and the number of elements in a queue. The size of the array is at most $m/2$. The number of entries is bounded by the number of interior nodes in $T$, because an interior node could correspond to an LP, but not an LNP. Since each interior node in $T$ has between two and five children, and a suffix tree built from a string of $m$ characters has $m$ leaves, then picking the worst case, when each node has only two children, $T$ will have at most $m$ nodes. Placing and traversing at most $m$ entries in at most $m/2$ queues can be done in linear time.

Steps 3-7 are removing an LNP and taking care of all the patterns that overlap the LNP. For this situation, we have the following lemma.

**Lemma 4.3.** Given an occurrence of an LNP of length $l$, GTAC removes all possible patterns that overlap that occurrence in $O(l)$.

**Proof.** GTAC removes the all the overlapping substrings for a given occurrence of an LNP in steps 4 to 7. During these steps, the $i$-ptr and $d$-ptr go down a few nodes in the tree, possibly a leaf is deleted, an internal node is
converted to a leaf, and an entry in a queue is deleted. The $i$ and $d$ pointers may go up a node before following a suffix link, and then begin dealing with the next suffix. Of these operations, the only one that may take more than constant time is when the $i$-ptr and $d$-ptr go down the tree a few nodes.

We will give an argument for the $d$-ptr, with the argument for the $i$-ptr being similar. While the $d$-ptr can travel as much as $2l + 1$ nodes to get to the leaf representing a single suffix, it’s amortized cost for dealing with all $3l$ suffixes is constant. When the $d$-ptr moves up one node, and over one suffix link, it loses height at most two nodes in the suffix tree. This is because the node depth of the suffix $o\alpha$ is at most one more that the node-depth of $\alpha$ and a pointer can travel from one node to the next in constant time [5]. So in order to look for $3l$ suffixes, it loses up to $6l$ nodes due to following suffix links, moves forward at most $5l$ characters to cover the all the overlapping substrings on $2l$ characters on either side of LNP. Thus GTAC moves forward at most $11l$ nodes to remove $3l$ suffixes. □

For a single iteration of step 3, say an LNP of length $l$ with $n$ occurrences is found. It takes $O(l)$ time to remove each one of them. After $n - 1$ of them are removed, a new rule is created and so the remaining occurrence of the LNP is converted to correspond to this rule. So to reduce the size of the original input by $O(nl)$ characters takes $O(nl)$, and the amount that gets removed can never exceed to original size of the input $m$, so this phase is $O(m)$ as well. Thus the theorem is proved. □

5 Implementation and Simulation Results

Other work in the area of estimating the entropy of genetic sequences have used the same benchmark sequences to compare their estimates. These standard sequences, (available at [12]) come from a variety of sources and include the complete genomes of two mitochondria: MPOMTGC, PANMTPCGA (also called MIPACGA); two chloroplasts: CHMTXX and CHMPXX (also called MPOPCPG); the complete chromosome III from yeast: SCCHRII (also called YSCCHRII); five sequences from humans: HUMGHCSA, HUMHBB, HUMHABCD, HUMDSTROP, HUMHPRTB; and finally the complete genome from the two viruses: VACCG and HEHCMVCG (also called HS5HCMVCG). On these test sequences, GTAC always beats Biocompress-2. As well, GTAC beats CDNA on eight out of the ten sequence results that are available for CDNA. The entropy estimates of all three algorithms are presented in Table 2.

When GTAC completely ignores reverse complements, the values are only slightly worse (about 0.01-0.02 bits/symbol) for eight of the twelve sequences, but dramatically different for four sequences: the two chloroplasts, CHMTXX and CHMPXX, and the two viruses, VACCG and HEHCMVCG. The results get worse by between 0.09 - 0.30 bits per symbol. This is because these sequences are known to contain long reverse complements. Two of these sequences are the ones that Biocompress-2 beats CDNA, and values were not available from CDNA for the other two.

We have performed some preliminary experiments with interesting consequences. The first result concerns coding and noncoding regions in E. coli. Around 90% of the genome of higher eukaryotes is noncoding whereas about 15% of the genome of E. coli is noncoding. If noncoding regions have a definite role, they may be more regular than coding regions which would support the conjecture that noncoding regions in prokaryotes are

<table>
<thead>
<tr>
<th>Sequence name</th>
<th>Sequence length</th>
<th>Unix compress</th>
<th>Bio-compress-2</th>
<th>CDNA compress</th>
<th>GTAC</th>
</tr>
</thead>
<tbody>
<tr>
<td>PANMTPCGA</td>
<td>106314</td>
<td>2.12</td>
<td>1.88</td>
<td>1.85</td>
<td>1.74</td>
</tr>
<tr>
<td>MPOMTGC</td>
<td>186609</td>
<td>2.20</td>
<td>1.94</td>
<td>1.87</td>
<td>1.78</td>
</tr>
<tr>
<td>CHMTXX</td>
<td>155844</td>
<td>2.19</td>
<td>1.62</td>
<td>1.65</td>
<td>1.53</td>
</tr>
<tr>
<td>CHMPXX</td>
<td>121124</td>
<td>2.09</td>
<td>1.68</td>
<td>-</td>
<td>1.58</td>
</tr>
<tr>
<td>SCCHRBI</td>
<td>315339</td>
<td>2.18</td>
<td>1.92</td>
<td>1.94</td>
<td>1.82</td>
</tr>
<tr>
<td>HUMGHCSA</td>
<td>66495</td>
<td>2.19</td>
<td>1.31</td>
<td>0.95</td>
<td>1.10</td>
</tr>
<tr>
<td>HUMHBB</td>
<td>73368</td>
<td>2.20</td>
<td>1.88</td>
<td>1.77</td>
<td>1.73</td>
</tr>
<tr>
<td>HUMHABCD</td>
<td>58864</td>
<td>2.21</td>
<td>1.88</td>
<td>1.67</td>
<td>1.70</td>
</tr>
<tr>
<td>HUMDSTROP</td>
<td>38770</td>
<td>2.23</td>
<td>1.93</td>
<td>1.93</td>
<td>1.81</td>
</tr>
<tr>
<td>HUMHPRTB</td>
<td>56737</td>
<td>2.20</td>
<td>1.91</td>
<td>1.72</td>
<td>1.72</td>
</tr>
<tr>
<td>VACCG</td>
<td>191737</td>
<td>2.14</td>
<td>1.76</td>
<td>1.81</td>
<td>1.67</td>
</tr>
<tr>
<td>HEHCMVCG</td>
<td>229354</td>
<td>2.20</td>
<td>1.85</td>
<td>-</td>
<td>1.74</td>
</tr>
</tbody>
</table>

Table 2: Comparison of Entropy Values in Bits Per Symbol
not junk. Our results confirmed this hypothesis. When comparing coding and noncoding regions of E. coli we found the following entropy values:

- 1.85 bits/symbol for coding regions (4,090,525 bases)
- 1.80 bits/symbol for noncoding regions (640,039 bases)

These results are consistent with Farach et al. [2] who did a similar experiment comparing the entropy of introns and exons from human sequences.

The second hypothesis we wanted to check was to verify if highly expressed essential genes have lower entropy than normal genes in E. coli because random mutations in normal genes are less likely to be deleterious. The results are as follows:

- 69 highly expressed essential genes: Mean: 1.7521 and sample variance: 0.0043 bits/symbol.
- 244 normal genes: Mean: 1.785 and sample variance: 0.0031 bits/symbol.

By statistical tests, with over 99% confidence, our hypothesis is supported.

6 Conclusion

While the idea of using Context Free Grammars in compression algorithms has been around for a while, the recent results have shown that if these grammars have the addition property that they are asymptotically compact then they are universal. This result has created a whole new family of approaches. One such algorithm in this family, namely GTAC, beats all known competitors for estimating the entropy of on a set of standard genetic sequences, and has the additional property that it has linear running time, and has been proven to be universal without assuming an ergodic source.

We are continuing to work in this area by modifying GTAC to include some of the approaches that other methods use, such as recognizing inexact matches which CDNA does.

7 Acknowledgments

We would like to thank Jonathan Badger, Paul Kearney and Huaiuchun Wang for helpful discussion and useful data.

References


